Partial application (""currying"")

Recall every function in ML takes exactly one argument.

Previously, we simulated multiple arguments by using one n-tuple argument.

Another way: take one argument and return a function that takes another argument and ...

This is called "currying" after its inventor, Haskell Curry

Example:

\[
\begin{align*}
\text{val inorder3 } & = \text{ fn x } \Rightarrow \text{ fn y } \Rightarrow \text{ fn z } \Rightarrow \\
& \text{ z } \Rightarrow \text{ y andalso y >= x} \\
\text{inorder3 } & \text{ 4 5 6} \\
\text{val is_pos } & = \text{ inorder3 } \text{ 0 0}
\end{align*}
\]

Currying vs. Pairs

Currying is elegant, but a bit backward: the function writer chooses which partial application is most convenient.

Of course, it's easy to write wrapper functions:

\[
\begin{align*}
\text{fun other_curry1 } f & = \text{ fn x } \Rightarrow \text{ fn y } \Rightarrow f y x \\
\text{fun other_curry2 } f x y & = f y x \\
\text{fun curry } f x y & = f \text{ (x,y)} \\
\text{fun uncurry } f \text{ (x,y)} & = f x y
\end{align*}
\]

More currying idioms

Currying is particularly convenient when creating similar functions with iterators:

\[
\begin{align*}
\text{fun fold_old } (f, \text{acc}, \text{l}) & = \\
& \text{ case l of} \\
& \quad \square \Rightarrow \text{acc} \\
& \quad \text{hd::tl} \Rightarrow \text{fold_old } (f, f(\text{acc,hd}), \text{tl}) \\
\text{fun fold_new } f & = \text{ fn acc } \Rightarrow \text{ fn l } \Rightarrow \\
& \text{ case l of} \\
& \quad \square \Rightarrow \text{acc} \\
& \quad \text{hd::tl} \Rightarrow \text{fold_new } f (f(\text{acc,hd})) \text{ tl} \\
\text{fun sum1 } l & = \text{ fold_old } ((\text{fn } (x,y) \Rightarrow x+y), 0, 1) \\
\text{val sum2 } & = \text{ fold_new } (\text{fn } (x,y) \Rightarrow x+y) 0
\end{align*}
\]

There's even convenient syntax: \(\text{fun fold_new } f \text{ acc l } = \ldots\)